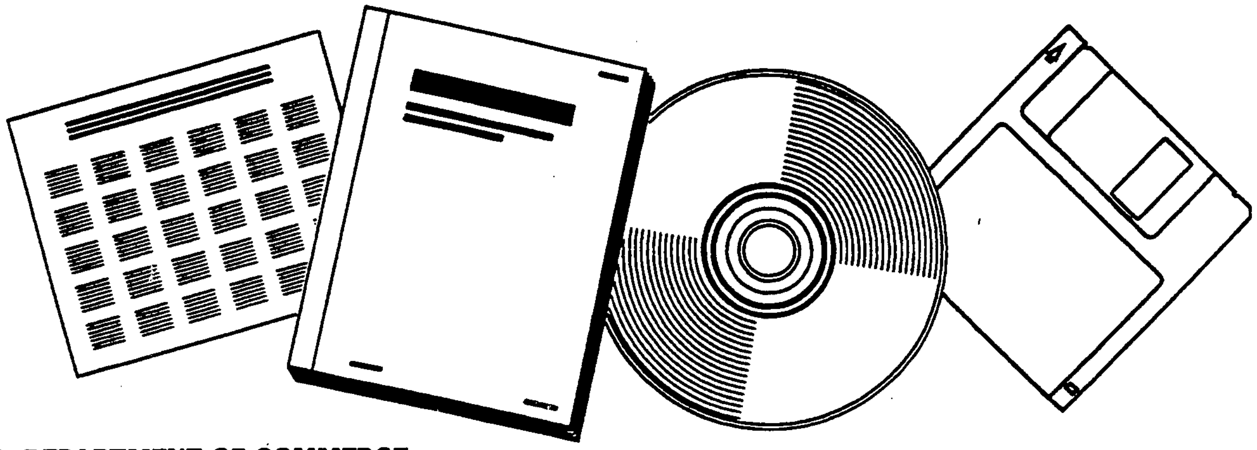
PARALLELIZING THE OPS5 MATCHING ALGORITHM IN QLISP

STANFORD UNIV., CA

OCT 91



October 1991

Report No. STAN-CS-91-1396

PB96-148556

Parallelizing the OPS5 Matching Algorithm in Qlisp

by

Daniel J. Scales

Department of Computer Science

Stanford University

Stanford, California 94305

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

REPORT DATE
October 1991

3. REPORT TYPE AND DATES COVERED

4. TITLE AND SUBTITLE

Parallelizing the OPS5 Matching Algorithm in Qlisp

5. FUNDING NUMBERS

6. AUTHOR(S)

Daniel J. Scales

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Computer Science Department
Stanford University
Stanford, CA 94305

8. PERFORMING ORGANIZATION REPORT NUMBER

STAN-CS-91-1396

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

DARPA
1400 Wilson Blvd.
Arlington, VA

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT ***(Maximum 200 words)***

In recent years, production systems have become a popular framework within which to implement large-scale expert systems. Unfortunately, production systems are often characterized by slow running times, because of the large amount of matching that must be done during their execution. For the production system language OPS5, there is a highly efficient matching algorithm known as the Rete algorithm which gives a large speedup over a naive implementation of production systems. In this paper, we describe our attempts to speed up OPS5 even further by parallelizing the Rete algorithm in Qlisp, a parallel Lisp language. We give details on the Qlisp constructs we used to parallelize the Rete algorithm and provide actual timing results on various OPS5 rule sets.

14. SUBJECT TERMS

15. NUMBER OF PAGES
20

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT

18. SECURITY CLASSIFICATION OF THIS PAGE

19. SECURITY CLASSIFICATION OF ABSTRACT

20. LIMITATION OF ABSTRACT

Parallelizing the OPS5 Matching Algorithm in Qlisp

Daniel Scales

Department of Computer Science
Stanford University
Stanford, CA 94305

October 1991

Abstract

In recent years, production systems have become a popular framework within which to implement large-scale expert systems. Unfortunately, production systems are often characterized by slow running times, because of the large amount of matching that must be done during their execution. For the production system language OPS5, there is a highly efficient matching algorithm known as the Rete algorithm which gives a large speedup over a naive implementation of production systems. In this paper, we describe our attempts to speed up OPS5 even further by parallelizing the Rete algorithm in Qlisp, a parallel Lisp language. We give details on the Qlisp constructs we used to parallelize the Rete algorithm and provide actual timing results on various OPS5 rule sets.

1 Introduction

In recent years, production (rule) systems have become a popular framework within which to implement large-scale expert systems. They provide a means of organizing a large amount of expert knowledge in the form of a collection of rules, each of which encodes a small bit of expert knowledge about a situation. Such an organization, in which most of the knowledge is stored declaratively in the form of many loosely connected rules (rather than, for instance, procedurally in the control flow of a program), makes it easier to add knowledge incrementally to an existing knowledge base, either by modifying existing rules or adding new rules to the rule system.

Unfortunately, production systems can run quite slowly, because of the large amount of matching needed to determine which particular rule applies at each stage in the execution of the system. Hence, any method of speeding up the matching process of rule systems is of great interest and utility. For a particular production system language, OPS5 [2], there is a highly efficient matching algorithm known as the Rete algorithm. However, even when this algorithm is used, a rule system with a fairly large number of rules may run unacceptably slowly. In this paper, we investigate parallelizing the Rete algorithm as a way to achieve greater execution speed for rule systems. In particular, we investigate parallelizing a Common Lisp implementation of the the Rete algorithm via the Qlisp parallel programming language. We base our work on the study by Gupta [6] of parallelism in the Rete algorithm. Additionally, Okuno and Gupta [10] have previously done some work investigating the use of Qlisp to speed up OPS5, using a Qlisp simulator.

Gupta et al. [7] describes the results of parallelizing an implementation of the Rete algorithm in C. Although a C implementation of the Rete algorithm is much faster than a Lisp implementation, it is interesting to speed up a Lisp implementation, because AI applications that use production systems are often built on top of Lisp. Additionally, the Rete algorithm provides an interesting "real-world" test case for Qlisp, since it is different in a number of ways, as will be described below, from the typical "toy" benchmark programs that are often used to test parallel Lisps.

2 OPS5

A production system consists of a collection of rules called *productions* and a working memory. Each production is specified by a set of conditions and a set of actions, such that the production is eligible to execute its actions ("fire") when its conditions are satisfied. Working memory is a global database containing data elements which are referenced by the conditions of the productions and created, modified, and removed by the actions of the productions. A production system interpreter executes a set of productions by repeatedly executing a cycle which consists of the following phases:

- *match phase* - determine all productions whose conditions are satisfied by the current contents of working memory.
- *conflict resolution phase* - choose a subset of the productions whose conditions are satisfied.
- *action phase* - execute the actions of those productions, possibly changing the contents of working memory in the process.

The set of all productions whose conditions are currently satisfied is called the *conflict set*. Thus, the second phase above consists of determining which of the productions currently in the conflict set should be executed.

OPS5 is a production system language in which all working-memory elements (*wme's*) are vectors of symbols (or numbers). The individual components of a wme are referred to as the *fields* of the wme. Typically, the value of the first field of a wme is interpreted as the *class* of the wme and the value of the remaining fields as attributes of the wme. OPS5 allows the user to assign symbolic names to the fields of wme's of each class. Such attribute names are indicated by a preceding up-arrow. When an attribute name appears in a condition or wme, it indicates that the following value or pattern refers to the field corresponding to that attribute. For example, the notation:

```
(goal ^status active ^type holds ^object ladder)
```

indicates a wme with class name "goal", a value of "active" for the "status" field, a value of "holds" for the "type" field, and a value of "ladder" for the "object" field.

A typical OPS5 production is as follows:

```
(p mb2
   (goal   ^status active  ^type holds  ^object <w>)
   (object ^name   <w>     ^at   <p>    ^on     ceiling)
   (object ^name   ladder  ^at   <p>)
-->
   (make goal ^status active ^type on ^object ladder)
)
```

The symbols `<w>` and `<p>` are names of OPS5 variables. A variable in a condition matches any value of the corresponding field. However, the variable is bound to the value it matches, and any other occurrence of the same variable within the conditions of the production matches only that value. Each wme that matches the condition is said to be an *instantiation* of that condition. For example, a wme is an instantiation of the first condition of the above production if and only if it has a class name of "goal", a value of "active" for the "status" field, and a value of "holds" for the "type" field.

An instantiation of a production is a list of wme's such that each wme of the list is an instantiation of the corresponding condition of the production, and all occurrences of each variable throughout the conditions can be bound to the same value. A particular instantiation of a production chosen to fire during conflict resolution is executed by performing each of the actions of the production, after replacing any variables in the actions by the values to which they were bound in the instantiation.

3 The Rete Algorithm

For OPS5, there exists an efficient serial algorithm, known as the Rete algorithm [1], for matching the conditions of the productions against working memory. The Rete algorithm builds a network representing the productions, called the Rete network, which is similar to a dataflow network. The Rete network takes advantage of two properties of production systems which allow for efficient matching:

- The contents of the working memory change slowly.
- There are many sequences of tests that are common to the conditions of more than one production.

The Rete algorithm exploits the first property by storing match information in the network between cycles, so that it only matches a wme against each production condition once, even if (as is likely) the wme remains in working memory for many cycles. Because of the storing of match information in the network, only the changes to working memory, rather than the whole contents of working memory, need be processed during each cycle. Each time a wme is added to working memory, the wme is "filtered" down the network, causing new partial matches to be recognized and recorded in the network and possibly causing one or more productions to be fully instantiated and placed in the conflict set. An identical process occurs when a wme is removed from memory, except that partial matches are discarded as the wme filters down the network. [1]

[1]This description is not completely accurate if any productions contain negated conditions, which "succeed" only if they are *not* matched by any element in working memory. Such negated conditions merely add

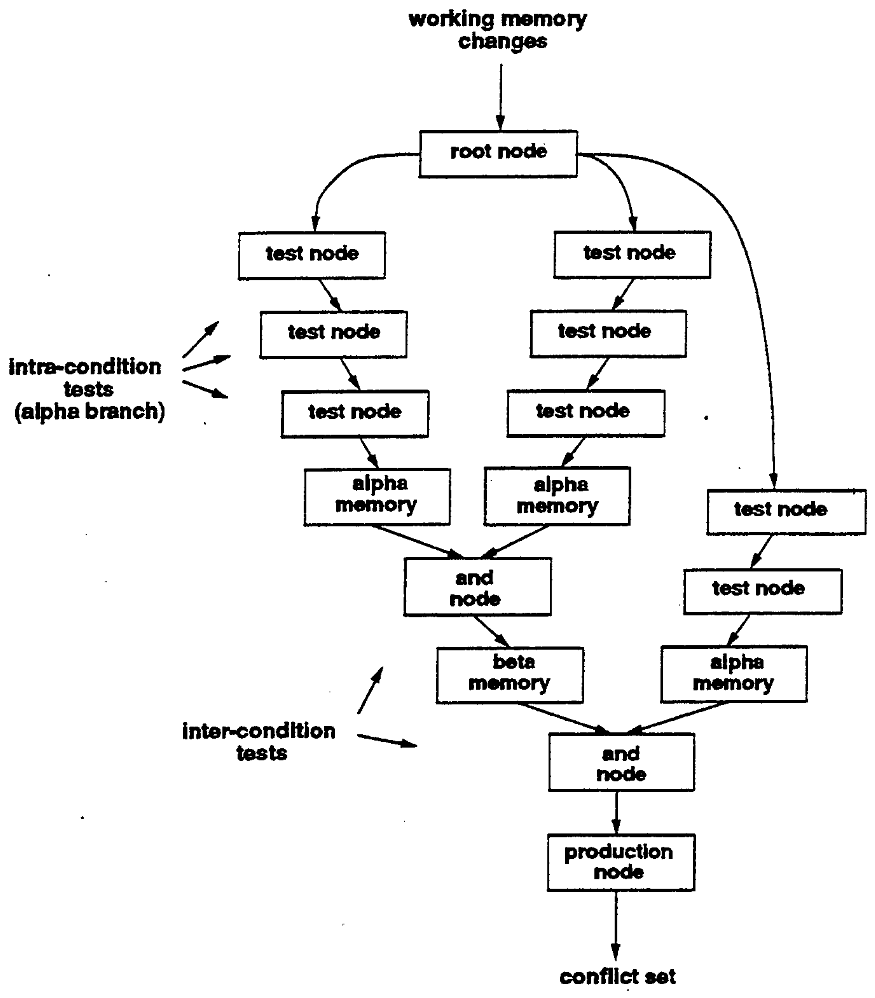


Figure 1: Structure of a Simple Rete Network

The Rete algorithm takes advantage of the second property by sharing common test and memory nodes as it adds nodes to the network to represent the conditions of each successive production. Because of the structure of the Rete network, the Rete algorithm can easily determine, as it is adding nodes to the network to represent a particular production, whether the nodes required already exist in the network and so can be reused.

Information flows among the nodes of the Rete network in the form of tokens, which are ordered lists of wme's. A match to a list of conditions (e.g. the left side of a production) is represented by a token in which the first wme matches the first condition, the second wme matches the second condition, and so on. A token stored at a node in the network represents a successful instantiation of the list of conditions represented by the set of nodes leading into that node.

The structure of a simple Rete network is displayed in Figure 1. Each condition of a

to the complexity of the Rete algorithm, without having a significant effect on the possibilities of parallelism, so we will not discuss them further.

production is represented in the Rete network by a singly-linked list of nodes, called an *alpha branch*. These nodes, referred to as *alpha nodes*, execute the *intra-condition* tests, those tests which concern only the wme that the condition is matching. For example, the intra-condition tests of the third condition of the production above are (1) the class name must be "goal" and (2) the "name" field must have the value "ladder". Each alpha branch is terminated by a memory node, called an *alpha memory*, in which is stored all tokens representing wme's that have satisfied all the tests of the alpha branch. Each alpha branch is linked to the root node of the network.

The two alpha branches representing the first two conditions of a production are joined together by an *and-node*, which has the alpha memories of the two branches as "inputs". A *beta memory* node is linked to the and-node as its output, storing any tokens which leave the and-node. If there is a third condition in the production, the alpha memory representing that condition is joined by another and-node to the beta memory representing the first two conditions, and again a beta memory is linked as the output of the new and-node. Similarly, the alpha memory of each successive condition of the production is joined by an and-node to the beta memory representing all previous conditions. The memory node at the left input of an and-node always represents the first n conditions of a production. while the memory node at the right input represents the $(n + 1)st$ condition. If a token from the left and right memories of an and-node have the same values for fields labeled by the same variables in the production, then they can be concatenated to form a new token which matches the first $n + 1$ conditions of the production. Hence, each and-node contains a list of the *inter-condition tests* which insure that the token from the left memory is consistent with the token from the right memory. A *production node* is linked to the final and-node that represents all of the conditions of a production. Any token that filters down to a production node represents a full instantiation of the corresponding production.

Whenever there is a change to working memory (either an addition or deletion of a wme), a token representing that change is created and sent down the network, starting at the root node. Any token reaching a memory node is stored in (or removed from) the memory before being sent on. If a token reaches an and-node, then it is matched against each token in the opposite memory of the and-node. If a token from the opposite memory is consistent with the newly-arrived token. as determined by the inter-condition tests, then the two tokens are combined, and the new token continues down the network. Whenever a token reaches a production node, an entry indicating that the token is an instantiation of the corresponding production is added to (or removed from) the conflict set. The processing that occurs at any node of the network when a single token reaches one of its inputs is called an *activation* of the node.

4 Qlisp

Qlisp is a parallel Lisp language proposed by Gabriel and McCarthy [3,4] based on a shared-memory processing model. It is an attempt to add a minimal set of high-level parallel constructs to Common Lisp in a consistent and useful way. The Qlisp constructs allow the creation of independent processes that can evaluate Lisp forms concurrently. All such processes have access to all data in memory. The 'Q' in Qlisp comes from the fact that all processes are assigned to processors by a run-time scheduler from a central queue (or set of queues), so that Qlisp programs can run on any number of processors without being rewritten or recompiled.

Goldman and Gabriel [5] describe an implementation of Qlisp on an Alliant FX/8 multiprocessor. This implementation includes the constructs originally proposed by Gabriel and McCarthy, as well as a number of other primitives to aid in building parallel Lisp programs. Additionally, Pehoushek [11] has implemented lower-cost versions of some of the basic constructs of Qlisp, which are useful in achieving better speedups in programs with finer-grain parallelism. Below, we describe the constructs that were most useful in this work. (For complete information on all Qlisp constructs, refer to Weening [12]).

In many of the constructs, a *proposition* is supplied as an argument which determines whether the construct will execute serially or in parallel. It is intended that the programmer can use the proposition to limit the creation of new processes at points in a computation when a large number of processes already exist.

4.1 QLET

qlet is a version of the **let** construct which may execute the bindings in parallel. It has the form:

(**qlet** *prop* ({*var binding-form*}*) {*main-form*}*)

That is, the **qlet** construct has the same form as the **let** construct, except for the addition of the proposition. If the value of *prop* is **nil**, then the **qlet** behaves in exactly the same way as the corresponding **let**, binding each variable to the value of its corresponding form, and then executing *main-form* in the environment with those new bindings. If *prop* evaluates to neither **nil** nor the special value **eager**, then a process is spawned corresponding to each variable, and each of the *binding-forms* are executed in parallel. When all of the processes executing the *binding-forms* have completed, each variable is bound to the value of the corresponding *binding-form*, and the main form is executed. Note that, in this case, the **qlet** enforces an implicit synchronization point, since the parent process must wait for the child processes to complete the evaluation of the binding-forms before continuing execution. Finally, if the value of the proposition is **eager**, then processes are spawned to evaluate each of the binding forms in parallel, but each of the variables is bound to a special data value known as a *future*, and the main form begins executing immediately. The main form executes in parallel with the evaluation of the binding forms, except that when, if ever, it references the value of one of the variables bound to a future, it waits until the process evaluating the "actual" value of the variable completes.

4.2 QLAMBDA

qlambda is a version of the **lambda** construct, which may cause execution of the lambda body in another process. It has the form:

(**qlambda** *prop* (*arg-list*) *body*)

qlambda creates a closure which is a critical region: only one process may be executing the closure at any given time. If *prop* evaluates to **nil**, then the closure is executed serially by the process that calls it. However, the calling process will be suspended on a queue associated with the **qlambda** if there is already another process executing the closure. Hence, a **qlambda** may be thought of as being implemented by a *sleep lock*, a type of lock which causes a process to be suspended (rather than just spinning) if it can not immediately acquire the lock. If the value of *prop* is not **nil** (and not **eager**), then the **qlambda** creates a separate process associated with the closure. A call to this *process closure* causes the **qlambda** process to begin execution of the closure with the indicated arguments, and a future to be returned immediately to the calling process. If the **qlambda** process is already executing another call, then any further calls are added to a queue associated with the process, to be executed one at a time in order. However, because futures are returned to the calling processes, they need not wait on the **qlambda** process until they actually require the values associated with the futures. A slightly different, but as yet unimplemented behavior is defined if the value of *prop* is **eager**, but we shall not describe it here.

4.3 SPAWN

The **spawn** construct has the form

(**spawn** *prop form*)
(**spawn** (*prop* **:for-effect t**) *form*)

If the value of *prop* is **nil**, then **spawn** just evaluates *form* normally (in the current process). If the value of *prop* is not **nil**, then a process is spawned to evaluate *form* and a future representing that form is returned, so that the current process can continue executing in parallel with the spawned process until it "needs" the value returned by *form*. If the future representing the spawned process is garbage-collected because there are no more references to it, then the spawned process will be killed, since its return value is no longer needed. If the form is being executed for side-effects rather than exclusively for its return value, then the keyword **:for-effect** should be supplied, in order to indicate that the process should not killed, even if its return value is ignored.

4.4 QWAIT

The **qwait** construct has the form

(**qwait** *form*)

qwait waits for all processes spawned during the evaluation of *form*, as well as all calls to **qlambda** process closures, to complete before returning the value of *form*. It is useful for waiting for the completion of processes which are running independently of their parent processes, because they were spawned "for effect" or because the futures created when they were spawned were never touched.

4.5 MAKE-LOCK, GET-LOCK and RELEASE-LOCK

The **make-lock**, **get-lock**, and **release-lock** constructs provide standard spin lock functionality. They provide lower-cost alternatives to the locking provided by **qlambda**.

4.6 QLET&, SPAWN&, and QWAIT&

An extension to Qlisp provides low-cost variants of **qlet**, **spawn**, and **qwait**, called **qlet&**, **spawn&**, and **qwait&**. These forms have much less overhead than the standard Qlisp constructs in creating, scheduling, and waiting for processes, but have some restrictions and slightly different behavior. Additionally, there is no eager version of **qlet&**.

The restrictions and the different behavior of the variants result from the different way in which they create and represent processes. In the standard Qlisp implementation, when a process must be created to evaluate a particular form `F`, a full closure `(lambda () F)` over the form is created to represent the process. This closure can then be evaluated by the processor that is eventually assigned the process. However, **qlet&** and **spawn&** represent a process in a much cheaper way as a list which consists of a function to be called and the value of each of the arguments with which it is to be called. [2] Since such a representation cannot be used for a process which is to evaluate a special form, **qlet&** and **spawn&** require that `F` be a function call `(fn f1 f2 ...)`, where `fn` is a function and `f1`, `f2`, ... are arbitrary forms. Specifically, all of the *binding-forms* of a **qlet&** expression must be function calls, and, similarly, the form within a **spawn&** must be a function call.

Additionally, this representation requires that the arguments `f1`, `f2`, ... to the function call be evaluated *before* the process is created, so the arguments are evaluated by the "parent" process (the process evaluating the **qlet&** or **spawn&**), rather than in the newly created "child" process. This change in the point at which the arguments are evaluated could obviously greatly reduce the amount of parallelism obtained if much of the work of the form `f` is in evaluating the arguments `f1`, `f2`, ... However, typically such a problem can be avoided by rewriting `F` as a a call to an auxiliary function `G` that takes as arguments just the free variables in the forms `f1`, `f2`, ... and does the work of evaluating the forms `f1`, `f2`, ... and then calling `fn` on the results.

The fact that the arguments of the form `F` are evaluated in the parent process rather than the child not only may change the degree of parallelism, but also can change the semantics of the form. Because the standard versions of **qlet** and **spawn** create full closures for processes, the child process shares with the parent process all lexical variables that it does not rebind locally. Thus, when it references one of these lexical variables, the child process will see any changes that have been made to the variable by the parent process (or other child processes) up to that point. However, the processes created by **qlet&** and **spawn&** do not share lexical variables with the parent, since no closure is created and all of the arguments in the form `F` are pre-evaluated in the parent process. Frequently, the latter behavior is actually the one that is more desirable. For example, the expression:

```
(dotimes (i 100) (spawn& (do-task i)))
```

evaluates the forms `(do-task 1)`, `(do-task 2)`, ..., `(do-task 100)` in parallel, as might be expected. However, if `spawn&` is replaced by `spawn`, then the above expression behaves unpredictably, because the hundred processes created all share the lexical variable `i`. In the worst case, if the parent process completes all the iterations of the loop before any of the child processes run, then each child process will execute `(do-task 100)`, because `i` will

[2] Also included in the representation of a process is a pointer to the special variable environment in which the process is created.

have the value 100 when they finally run. For the processes to behave as intended, each must be given a private copy of the index variable `i`, as follows:

```
(dotimes (i 100)
        (let ((i1 i)) (spawn (do-task i1))))
```

5 Parallelism in the Rete Algorithm

Gupta [6] describes several types of parallelism that might be exploited in speeding up the conflict resolution and action phases of a rule system. However, as might be expected, the time spent in the matching phase usually greatly dominates the processing time of a production system. Hence, in this work, we only investigate parallelizing the match phase of a production system. In this section, we describe several levels of parallelism in the Rete algorithm, give some examples of how the Qlisp constructs can be used to parallelize the algorithm, and then summarize some of the interesting aspects of the parallel Rete algorithm.

5.1 Levels of Parallelism in the Rete Algorithm

As described above, in the Rete algorithm, each change to working memory causes a flow of tokens throughout the Rete network, representing changes in partial matches of productions caused by that particular change to working memory. At some of the nodes of the network, there will be much activity, as many tokens are generated and sent on due to a single change to working memory, whereas at other nodes there will be no work, because no new tokens reach them. As identified by Gupta [6], there are several levels of potential parallelism in this filtering process of the Rete algorithm. These are:

- *rule-level parallelism* - doing the work for the parts of the network representing different rules in parallel.
- *node-level parallelism* - doing the work at the individual and-nodes in parallel.
- *intra-node parallelism* - doing the work of each activation of each and-node in parallel.

The types of parallelism above are listed in order of decreasing task granularity. As usual, there is a tradeoff between the greater possible parallelism achievable with smaller task granularity vs. the increased scheduling overhead and contention.

Additionally, we can introduce further parallelism into each of these schemes via *change* or *action parallelism*, in which the filtering process is done for many changes to working memory in parallel. All of the changes resulting from the firing of a production in the execution phase of one cycle can be processed in parallel during the match phase of the next cycle. We can expose even more parallelism if we allow the action and match phases to overlap somewhat, by beginning the filtering process for a change to working memory as soon as the change is generated by an action. The only requirement for correctness (as with all types of parallelism mentioned so far) is that, on each cycle, all processes involved in the matching phase complete before the succeeding conflict resolution phase commences.

5.2 Details of Parallelizing the Rete Algorithm in Qlisp

We have implemented and investigated all three levels of parallelism described above, both alone and in conjunction with change parallelism. Below, we give some details and some specific examples of parallelizing the Rete algorithm in Qlisp. For full details on implementing intra-node parallelism, the level which gives the best speedup. refer to Gupta et al. [7].

The basic parallelism of the algorithm comes from the structure of the network. The root node of the network, at which all changes to working memory start, has many outputs, one for each distinct condition that any change to working memory must be matched against. Additionally, due to the similarity of many of the conditions of different rules, many other nodes in the network are shared between several conditions and/or rules and so have multiple outputs. Parallelism is created by spawning a different process to handle the flow of a token to each one of the outputs of a node. In Qlisp, this procedure might be coded as:

```
(defun send-to-outputs (token outputs)
  (qlet t ((x (eval-node token (car outputs)))
           (y (send-to-outputs (cdr outputs)))))
```

or

```
(defun send-to-outputs (token nodelist)
  (dolist (node nodelist)
     (let ((n node))
        (spawn (t :for-effect t) (eval-node token n)))))
```

In the first example, we have used a recursive **qlet** construction (with no actual body) to create a variable number of processes, each of which sends **token** to a different output. Depending on the Qlisp implementation, this construction might be quite inefficient, since it creates twice as many processes as there are outputs, with half the processes doing real work and half the processes (the ones that execute the **qlet**) merely waiting for their child processes to complete.

In the second example, we have used **spawn** to create exactly as many processes as are needed. We have created the processes with the **:for-effect** keyword, since they are executed for effect and should not be garbage-collected, even though the futures they return are ignored, Most often, such **spawns** should be executed within the dynamic scope of a **qwait** call, which establishes a barrier requiring that all processes created within the **qwait** call complete before the **qwait** call returns. Since all of the processes spawned in the Rete algorithm are independent, only a single **qwait** call is needed "at the top". If the **qwait** is placed around the function that does the match for a single wme, then all processing associated with each change to working memory will be done in parallel. If, however, the **qwait** is placed around a "higher-up" function that does the match for all working memory changes in each cycle, then all processing in the network associated with all the working memory changes of a single cycle will occur in parallel, thus producing the change parallelism described above.

Though fewer processes are created in the **spawn** construction above than in the **qlet** construction, it is still possible that the **qlet** expression is cheaper, if creating a process via a (non-eager) **qlet** is quite a bit less expensive than creating one via **spawn**. This is quite possible, depending on the implementation, since a spawned process must be completely independent (in terms of stack, control structures, etc.) of the process that created it. In contrast, much of the information about a process created by a (non-eager) **qlet** can be maintained by the parent process (perhaps on its stack), since the parent process must wait for the child process to complete.

qlet& and **spawn&** exhibit just such a cost tradeoff: the parallelism created by the **spawn&** construct is more flexible than the parallelism of the **qlet&** construct, but creating a process via **qlet&** is significantly cheaper than creating one via **spawn&**. Also, **qlet&** adds the extra efficiency of having the parent process itself execute one of the binding forms, thus entirely avoiding the creation of any extra processes. (This is similar to the way that an expression like `(+ (future A) (future B))` in Multilisp can be optimized to `(+ (future A) B)`, given a left-to-right evaluation of arguments [8].) Because of this tradeoff, it is sometimes not obvious whether to use **spawn&** or **qlet&** when creating processes.

Another important source of parallelism in node and intra-node parallelism occurs at the and-nodes. In contrast to the alpha nodes, the processing resulting from a single activation of an and-node can result in the output of several tokens, since the incoming token may cause the partial production represented by the and-node to match in several different ways. Hence, another source of parallelism may be exploited by processing the multiple tokens produced by a single activation of an and-node in parallel. The Qlisp construction to implement this parallelism looks something like this:

```
(defun and-node-left-activation (left-token binding-tests right-memory outputs)
    ...
    (dolist (right-token right-memory)
        (if (bindings-match left-token right-token binding-tests)
            (spawn t
                (send-to-outputs (concatenate left-token right-token) outputs))))
    ...
)
```

Here, an activation resulting from a token coming to the left input of an and-node is handled by matching the incoming token with each token in the right memory, and for each match, spawning a process which sends the concatenation of the two matching tokens to each of the outputs of the and-node. Note that here there is no corresponding **qlet** expression that could achieve the same effect of conditionally spawning off independent processes at varying intervals during a computation. Note also that, depending on the type of locking done by the and-node activation (as discussed below), this implementation may result in deadlock if run in serial mode (in which the spawn call does not create a new process), since the and-node activation may hold some lock that will be required by a later and-node activation caused by one of the tokens that is sent out. An alternate implementation that does not have this deadlock problem in serial mode is to collect all the tokens to be sent out and release any locks being held before sending out any tokens. Obviously, however, such an implementation reduces the amount of parallelism.

With all these processes filtering tokens through the network, it is, of course, a requirement to lock the shared data that may be accessed by the processes: the memories of the memory nodes and the conflict set. Both of these shared data structures can be "protected" by encapsulating the code that manipulates them in **qlambda** closures. As stated above, **qlambda** closures are critical regions which execute calls by processes serially, either within the calling process or in a separate process. For rule parallelism, there should be a separate **qlambda** closure for each group of memory nodes representing a single production. For node parallelism, there should be a separate **qlambda** closure for each and-node and the two memory nodes at its inputs. For intra-node parallelism, locking is even finer – through the use of a global bucket hash table to store the contents of the memory nodes – in order to allow several activations of a single and-node to run concurrently. In this case, there should be one **qlambda** closure per bucket of the hash table. Additionally, for all three levels of parallelism, there should be a single **qlambda** process for accessing the conflict set atomically. For more details on implementing the necessary locking for each level in Qlisp, refer to Okuno and Gupta [10].

Typically, in the Rete algorithm, one would most often want to use parallel **qlambda** closures, in order to allow the calling process to proceed immediately without waiting on a lock. However, a parallel **qlambda** has costs in terms of communication and context switching between the calling process and the process attached to the **qlambda** closure. If it expected that, most often, the wait for a lock on a shared data structure will be minimal, then it may be cheaper to achieve the locking required via the use of spin locks rather than calls to **qlambda** closures.

5.3 Properties of the Parallel Rete Algorithm

By way of a summary, we here list some of the characteristics of the Rete algorithm that distinguish it from many of the small, functional programs that are often used as test cases for parallel Lisps:

- When run in parallel, the algorithm requires extensive use of locks at nodes in the network that store state information. Hence, processes may not always be able to run to completion once they have started (if sleep locks are used) or may spend some of their processing time spinning (if spin locks are used).

- Although a correctly-implemented parallel version of the Rete algorithm will yield the same overall result for a particular set of working memory changes as the serial version, it will not necessarily process the same number of node activations, since the actual work done in the network depends on the order in which tokens are sent down the network.

- The algorithm is highly non-functional; most functions do not return values explicitly, but instead modify global variables and/or information stored in the network. Hence, with the proper locking, most processes that are spawned can run independently, with just a single synchronization point at the end of the matching phase of each cycle.

- The algorithm provides ample opportunities for parallelism, but the degree of parallelism at different points in the run and the size of the tasks created varies widely and unpredictably. Because of this variability, it is necessary to spawn processes at a fairly fine-grained level in order to achieve a reasonable speedup.

6 Results and Analysis

In Table 1, we give the timings results for our best parallel implementation of the Rete algorithm in Qlisp, which includes a combination of intra-node and change parallelism. Our implementation of intra-node parallelism was clearly faster than our implementations of rule- and node-level parallelism, as was expected, given the smaller grain size of its tasks. However, because of the small size of the tasks created at the level of intra-node parallelism, the use of the low-cost primitives **qlet&** and **spawn&** was crucial in keeping the overhead low enough to get reasonable speedups. The best times resulted from the use of **qlet&** to create parallelism at the outputs of the root and alpha test nodes and the use of **spawn&** to create the parallelism at the outputs of the and-nodes, as described in Section 5.2. Additionally, we used spins locks throughout rather than **qlambda**, which is quite expensive and has no low-cost equivalent. At all three levels of parallelism, the use of change parallelism improved performance significantly. The fact that change parallelism is clearly beneficial at all levels probably results from the fact that the individual changes to working memory made during a cycle of the production system often affect different productions and different nodes in the network, so they can be processed in parallel with little contention.

We began with the standard Common Lisp implementation of OPS5, but, before attempting to parallelize it, made a number of changes to the lowest-level functions involved in matching, in order to eliminate a number of inefficiencies. We also included type declarations in some of these functions in order to speed them up further. Additionally, in all implementations, both serial and parallel, we used the highest level of optimization of the Qlisp compiler (which is based on the Lucid Common Lisp compiler).

We also changed the implementation of the conflict set from a linked list to a hash table in the serial and parallel implementations. Several of the OPS5 programs that were used for timings typically cause runs in which the average size of the conflict set is very large. Hence, to reduce the total time adding and deleting instantiations from the conflict set (all of which must be serialized via a lock), the conflict set was converted to a hash table. This change does not measurably increase or decrease the times of those runs in which the average conflict set size is small.

All times are user CPU time in milliseconds and were obtained for an implementation of Qlisp on an eight-processor Alliant FX/8. The Alliant FX/8 is a shared-memory multiprocessor in which all of the processors are connected to a common global memory via a shared cache. The individual processors of the Alliant run at about three MIPS, though, some appear to run slightly slower when they are running in parallel mode, so the best speedup that can be obtained for 8 processors is about 7.8. Since the current implementation of Qlisp does not have a parallel garbage collector, all times are for runs in which there was no garbage collection.

For the timings, we use four OPS5 programs (rule sets) also used in [7] and [9]. They are:

program	no. of	no. of	RHS	average	serial	parallel time		speedup	
	rules	cycles	actions	CS size	time	1 procr	8 procr	"true"	"nom."
Rubik	70	50	1166	2	30937	38355	7138	4.33	5.37
Weaver	637	150	469	8	12148	33431	7344	1.65	4.55
Tourney	17	30	81	625	15675	11550	4866	3.22	2.37
Tourney*	17	30	81	625	14263	10138	3454	4.13	2.94
Waltz	33	535	2086	95	108977	74127	23431	4.65	3.16
Waltz*	33	535	2086	95	103876	69026	18330	5.66	3.77

*times excluding the serial conflict resolution phase

Table 1: Times (in ms) for the runs of four OPS5 programs

- **Rubik**, a program that models manipulations of a Rubik's cube
- **Weaver**, a VLSI routing program
- **Tourney**, a program that makes schedules for a tournament
- **Waltz**, a program that interprets three-dimensional line drawings

The first column of Table 1 gives the number of rules in each program. The second and third columns gives the number of cycles (rule firings) and the number of actions (additions and deletions to working memory) in the runs timed for each program, while the fourth column gives the average size of the conflict set over all cycles. The fifth column gives the time for the run of a rule set on the optimized serial implementation of OPS5. The sixth and seventh columns give the times for the run on the parallel implementation of OPS5 running on one or eight processors, respectively. For the **Tourney** and **Waltz** rule sets, the time for conflict resolution (which runs completely serially in our implementation) is such a significant part of the run time (about 1400 ms and 5100 ms respectively) that we include entries showing the results when the time for conflict resolution is excluded. For the other two rule sets, the time spent in the conflict resolution phase is basically insignificant (approximately 300ms and 350ms, respectively, for **Rubik** and **Weaver**).

The "true" speedup is the ratio of the time for the serial implementation to the time for the parallel implementation on eight processors. The "nominal" speedup is the ratio of the times of the parallel implementation on one and eight processors. The differences in the performance for each rule set of the serial implementation and the parallel implementation on one processor result mainly from two factors:

- Changes to the network required for the parallel implementation that decrease sharing and so increase the number of nodes in the network.
- Speedup in the access to tokens in the memories of memory nodes because of the use of hash tables in the implementation of intra-node parallelism.

The first item is quite significant and is the principal reason why **Weaver** runs nearly three times slower under the parallel implementation on one processor than under the serial implementation. The slow-down comes mainly from the fact that the implementation of

program	par. time	idle time	spin time	no. of procs. qlet&	spawn	average activ. time
Rubik	7138	1232	279	78926	7399	0.265
Weaver	7344	1402	480	62166	3078	0.390
Tourney	4866	2038	1145	1075	3195	6.634
Waltz	23431	8655	3033	87644	6949	0.954

Table 2: Statistics for parallel runs of four OPS5 programs

intra-node parallelism does not allow, in most cases, the sharing of memory nodes Thus, if an alpha memory is shared in the serial implementation by three rules (because all three rules contain an identical condition), then in the parallel implementation, that memory node must be duplicated three times, and the work that had been done by that one node is duplicated at each of the two extra nodes. This effect can become huge if many conditions and sequences of conditions are shared among rules. Note, however, that any extra work created because of the loss of sharing can probably run in parallel with the original work with little contention, since the extra work occurs at new nodes that didn't exist in the serial network. Hence, the effects of loss of sharing can be mitigated somewhat by applying more processors to the matching, if they are available.

The second factor is the reason that **Tourney** and **Waltz** actually ran faster on the parallel implementation than on the serial implementation. As mentioned above, part of the modifications required to implement intra-node parallelism involve changing the memories at the memory nodes from simple lists to hash tables – thereby changing the granularity of locks that must be acquired to modify the contents of a memory node from the level of the entire memory list to the level of a single bucket of a hash table. If, for a particular run of a set of rules, the number of tokens at various memory nodes becomes very large, then the processing of the memory nodes may be significantly faster when hash tables are used. Such is the case with the **Tourney** and **Waltz** programs. However, as indicated above, the use of hash tables at the memory nodes contributes to the loss of sharing and so does not always improve performance (as with the **Weaver** program).

Table 2 gives further statistics on the eight-processor parallel runs of the four OPS5 programs. The first column repeats the figures given in Table 1 for the running time on 8 processors. The second column gives the average idle time per processor during the run. This figure is the time that a processor is idle because no process is available to execute; it does not include the time spent spinning on locks in order to access token memories or the conflict set, which is the value given in the third column. The fourth and fifth columns give the number of processes created via **qlet&** and **spawn&** respectively. The last column indicates the average time for the processing of an and-node activation (in milliseconds) for each of the programs. This last figure gives an indication of the grain-size of the processes created. (However, these figures do not give the whole picture, since they do not average in the times for the processes that handle the tests on the alpha branches.) Obviously, the average activation time for the **Tourney** program is huge; such huge times cause load-balancing problems and increase the contention for locks associated with the memories of the and-nodes.

For purposes of comparison, it is interesting to note the approximate times required for

operation	time
setq, car, cdr	$1\mu s$
get-lock	$3\mu s$
cons	$5\mu s$
function call	$9-12\mu s$
creating a closure	$55\mu s$
creating a process via **qlet&**	$23\mu s$
transferring a process	$30\mu s$
suspending a process	$30\mu s$
creating a process via **spawn&**	$100\mu s$

Table 3: Times for some Qlisp operations on the Alliant FX/8

various operations in Qlisp on the Alliant, as given in Table 3. The figure for **cons** does not include any garbage-collection overhead that may be associated with the **cons**. The last four entries describe the overheads associated with the low-cost versions of the Qlisp constructs. The entry "transferring a process" gives the time involved in transferring a process from the processor on which it was created to another processor which is looking for work. The entry "suspending a process" is overhead involved in suspending a process because it must wait at a **qwait&** or for a sleep lock to be released or for the child processes of a **qlet&** to complete. The overhead in creating a process via **spawn&** is higher than the overhead in creating a process via **qlet&**, because more information must be maintained and many process structures must heap-allocated in order to allow a spawned process to run independently of its parent process.

We may summarize the results of the four different programs as follows:

- **Rubik** gave the best nominal speedups of the four rule sets. Not only do runs of this rule set have a good degree of parallelism, but the parallelism that is exploited does not generate much contention for memory node locks. Although many of the parallel tasks generated by a run of this rule set are quite small, as evidenced by the figure of 265 microseconds in Table 2 for the average time for an activation of an and-node, the **spawn&** and **qlet&** constructs are cheap enough that good speedup can still be achieved.

- **Weaver** achieves a rather poor "true" speedup. However, the nominal speedup is quite good, indicating that the poor speed is due to loss of sharing, rather than lack of parallelism or contention on locks. Because much of the processing power of the eight processors is used just in making up for the loss of sharing, it is not possible to tell the extent of the real parallelism in the rule set without using more processors.

- **Tourney** and **Waltz** show reasonably good "true" speedup, which is somewhat misleading, since much of the true speedup of the **Tourney** and **Waltz** programs results from the change of node memories from linked lists to hash tables, rather than from parallelism. In terms of parallelism, what is more important is their nominal speedup, which is rather poor. As explained in [7], the poor performance of **Tourney** is due to the fact that the structure of the small number of rules is such that a combinatorial number of partial matches to many of the rules is generated during the run. This causes large numbers of tokens to be stored in some of the memory nodes. and consequently, the average time to process memory node and and-node activations to become very large (as indicated by the huge figure in the "average activation time" column of Table 2). The increased processing time for node activations increases lock contention (the "spin time") and limits the degree of parallelism. **Waltz** appears to suffer from a similar problem, though to a much lesser extent, as evidenced by the large average conflict set size, large average activation time for and-nodes, and the large spin time per processor (as a percentage of the total run time).

7 Comments on Qlisp

In general, we found the Qlisp parallel constructs both reasonable and useful. Their similarity to existing Lisp constructs made them easy to remember, and their behavior and interaction with other parallel constructs was fairly easy to understand. In addition, parallelism was easy to add and remove in a Qlisp program, either by explicitly adding or removing a Qlisp construct (e.g. a **spawn** surrounding a particular form) or by changing the proposition associated with a Qlisp construct. This property of Qlisp facilitated experimentation with different kinds of parallelism. On the other hand, because the current Qlisp environment provides no tools to help the user in determining where to introduce parallelism in his program, the user may in fact have to do much experimentation to determine what combination of parallelism gives the best results.

The very different behavior of the non-eager **qlet** and **spawn** may cause some confusion. Whereas **spawn** either uses futures for synchronization or implies no synchronization at all (when specified as "for effect"), a non-eager **qlet** synchronizes via a barrier that requires that all child processes be completely finished before the parent process continues. Such difference in behavior means that the non-eager **qlet** and **spawn** are often best used in quite different circumstances, even though they are both constructs for creating parallelism.

spawn and **qwait** were quite useful in parallelizing the Rete algorithm, because of the non-functional and independent nature of the tasks created by the parallel versions of the algorithm. **spawn** was especially crucial to avoid the overhead of the many extra unneeded synchronization points that would result if **qlet** were used instead. In addition, **spawn** was useful at points in the algorithm where the number of tasks to be created was not fixed. As illustrated in Section 5.2 above, creating a variable number of tasks via **spawn** is straightforward, but, if **qlet** is used, requires defining a recursive function. Also, if the variable number of tasks are created over a period of time (as in `and-node-left-activation` in Section 5.2), then a **qlet** is unusable unless the potential tasks are collected and spawned all at once (which, of course, results in a loss of some of the parallelism). The **qwait** construct made it quite convenient to use **spawn** freely; any less flexible construct that required the programmer to keep track of exactly which processes or how many processes were to be waited on would make **spawn** much less useful.

qlambda mapped naturally to the locking needed in several parts of the algorithm, though spin locks were eventually used instead, because they have lower overhead. For completeness, it might be reasonable to add to Qlisp a version of **qlambda** based on spin locks.

One big problem with the current implementation is the lack of good performance monitoring tools. Common Lisp itself defines no tools other than the **time** construct for profiling the execution of a serial program. and Qlisp provides no additional tools for monitoring the execution of parallel programs. Additionally, on the Alliant FX/8, the only time that can be obtained with a low enough overhead to be useful in timing operations at the level of node activations is wall-clock time. Given the existence of other users and system processes on the machine, wall-clock time is obviously not sufficient for getting highly accurate timings. Hence, it is quite difficult to get an accurate profile of the timing characteristics of the parallel OPS5 algorithm running under Qlisp and so to determine where the bottlenecks are.

8 Conclusion

We may briefly summarize some conclusions from this work as follows:

- Qlisp is a useful extension to Common Lisp for parallelizing the expensive parts of computation-intensive Lisp programs. In our case, it was easy to parallelize the expensive matching phase of the OPS5 interpreter, while leaving untouched the remaining code that handles the building of the Rete network and the processing required for other two phases of the execution cycle.

- The basic Qlisp constructs are simple to understand, but powerful enough that it is easy to experiment with different kinds of parallelism.

- **qwait** and **spawn** are especially useful for programs which are highly side-effecting and whose opportunities for parallelism vary greatly at runtime, both of which are true of the OPS5 implementation of the Rete algorithm. In contrast, Qlisp's **qlet** construct is often sufficient and more efficient than **spawn** in many cases in which parallelism is more regular. We needed **spawn** to express parallelism at the and-nodes, but **qlet** was adequate and cheaper for expressing parallelism at the outputs of the alpha test nodes

- The standard Qlisp constructs (in the current implementation) have a fair amount of overhead and are intended to be used in implementing programs with medium-grained parallelism. However, the low-cost variants to the basic Qlisp constructs are cheap enough that programs with finer-grained parallelism can achieve reasonable speedups. This result is significant, since many of the symbolic applications for which Lisp is typically used have an irregular enough execution behavior that parallelism can only be exploited effectively at the fine-grain level.

- In particular, using the low-cost variants, we were able to obtain moderate speedups for runs of the parallel Rete algorithm on several different rule sets.

9 Future Work

There is a variety of work that can still be done:

- Investigate other OPS5 programs, in order to determine whether most programs yield good speedups, as with the **Rubik** program, or exhibit poor speedup, because of some of the problems described above in association with the **Weaver** and **Tourney** rule sets.

- Investigate the use of dynamic spawning [11], in which the proposition associated with most Qlisp constructs is used to limit the number of processes being created. We did not use the proposition for dynamic spawning, because it was not obvious how to determine how much work a particular task that might be spawned as a separate process will entail. Some limited tests in which spawning was controlled by the current size of each processor's task queue either increased or did not significantly change overall run times. Dynamic spawning may not be particularly relevant for the parallelized Rete algorithm, since the amount of usable parallelism is already highly limited by the variability of task sizes and by contention for locks.

- Investigate how easy it is to express in Qlisp the parallelism in the conflict resolution and action phases of OPS5.

- Investigate parallelizing other matching algorithms proposed for OPS5, such as the Treat algorithm [9], in Qlisp.

10 Acknowledgments

Thanks go to Joe Weening and Dan Pehoushek for comments on drafts of this paper. Thanks also to Dan Pehoushek for his willingness to extend the low-cost primitives to meet my needs in parallelizing the Rete algorithm and for his help and advice in using the primitives. This work was supported in part by the Defense Advanced Research Projects Agency under contract N00039-84-C-02111.

References

1. Forgy, C.L. *On the Efficient Implementation of Production Systems.* Ph.D. Thesis, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, 1979.

2. Forgy, C.L. *OPS5 User's Manual.* Technical Report CMU-CS-81-135, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, July, 1981.

3. Gabriel, R.P. and McCarthy, J. "Queue-based Multiprocessor Lisp," *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, ACM, Austin, Texas, August, 1984.

4. Gabriel, R.P. and McCarthy, J. "Qlisp," *Parallel Computation and Computers for Artificial Intelligence*, edited by Janusz S. Kowalik, Kluwer Academic Publishers, Boston, Massachusetts, 1988.

5. Goldman, R. and Gabriel, R.P. "Qlisp: Parallel Processing in Lisp," *Proceedings of HICSS-22, Hawaii International Conference on System Sciences*, January, 1989.

6. Gupta, A. *Parallelism in Production Systems.* Ph.D. thesis, Technical Report CMU-CS-86-122, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, March, 1986.

7. Gupta, A., et. al. "Parallel Implementation of OPS5 on the Encore Multiprocessor: Results and Analysis," *International Journal of Parallel Programming*, Vol. 17, No. 2, April, 1988.

8. Halstead, R.H., Jr. "An Assessment of Multilisp: Lessons from Experience," *International Journal of Parallel Programming*, Vol. 15, No. 6, December, 1986.

9. Miranker, D.P. *TREAT: A New and Efficient Match Algorithm for AI Production Systems.* Pitman/Morgan Kaufmann, 1989.

10. Okuno, H. and Gupta, A. *Parallel Execution of OPS5 in QLISP,* Technical Report STAN-CS-87-1166, Department of Computer Science, Stanford University, Stanford, June, 1987.

11. Pehoushek, J.D. and Weening, J.S. "Low-cost Process Creation and Dynamic Partitioning in Qlisp." *Proceedings of the 1989 US/Japan Workshop on Parallel Lisp.* Springer-Verlag (to appear in 1990).

12. Weening, J.S. *Qlisp Reference Manual.* In preparation.